Interval Scheduling: Fine-Grained Code Scheduling for Embedded Systems *

Pai Chou, Gaetano Borriello
Department of Computer Science and Engineering, Box 352350
University of Washington, Seattle, WA 98195-2350

Abstract – A central problem in embedded system co-synthesis is the generation of software for low-level I/O. Scheduling still remains a manual task because existing coarse-grained real-time scheduling algorithms are not applicable: they assume fixed delays even though the run times are often variable, and they incur too much overhead. To solve this problem, we present a new static ordering technique, called *interval scheduling*, for meeting general timing constraints on fine-grained, variable-delay operations without using a run-time executive.

I Introduction

One of the main goals of hardware-software co-synthesis is enabling the designer to quickly explore the design space. A co-synthesis tool maps a design to different architectures while meeting design requirements. One such requirement is timing. To meet timing constraints on a set of operations, we need to first estimate the *execution delays*, and then compute a *schedule* for the operations.

The execution times of the operations are often variable, rather than exact values. This uncertainty comes from pipelining, interrupts, and caching effects at the instruction level, as well as data dependency and control flow at the program level. Although timing prediction is an undecidable problem in general, it is still possible to compute execution time bounds in many practical cases [1]. To obtain tighter time bounds, timing prediction tools make use of user-annotated information such as loop count or mutually exclusive paths.

Existing scheduling techniques consider only one delay parameter, namely the *worst-case* delay, rather than an interval. Variations in execution delays do not pose a problem for coarse-grained scheduling algorithms, because they assume there is a run-time executive, which gains control on regular clock interrupts and dispatches operations at the appropriate times (Fig. 1). The executive is assumed to incur negligible overhead.

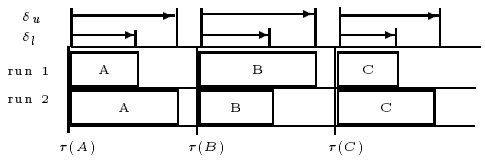

Fig. 1: Traditional model: if the execution delays of the operations A, B, and C can vary, the schedule is computed based on the worst-case delays. This relies on a run-time executive to start the operations at the same times in every run.

Unfortunately, these assumptions are not valid for fine-grained software operations, such as those that perform low-level I/O. These operations range from basic blocks to subroutines with hundreds of instructions. When the granularity is this small, a run-time executive will incur too much overhead and should be eliminated at this level. Our solution is to compute an *interval schedule*, which is a static ordering of the operations interleaved with fixed amounts of spacing (Fig. 2). A run-time executive may be present *between different runs* to handle timing constraints at a higher level, but it is assumed not to have control during the execution of interval schedules. An interval schedule is valid if for all combinations of actual delay values, all timing constraints are satisfied. This may be classified as *self-timed* scheduling according to [2] in the sense that resource assignment and ordering are statically determined, but the actual timing is not known until run-time.

The contribution of this paper is two fold. First, we present an exact algorithm for finding a valid interval schedule whenever one exists. We prove the algorithm's correctness in the appendix. Second, we study the effec-

*This work was supported by ARPA N00014-J-91-4041.

32nd ACM/IEEE Design Automation Conference ®

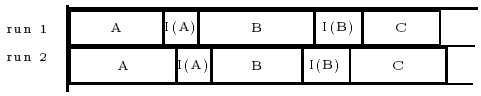

Fig. 2: Our model: we order the operations and compute the idle spacing $I(A)$ and $I(B)$ between them. The actual execution delays of the operations can vary from run to run, but $I(A)$ and $I(B)$ are fixed. This model does not require a run time executive.

operation	exec. delay		from the	to the	constraint	
p_i	$\delta_l(p_i)$	$\delta_u(p_i)$	start of	start of	min	max
A	1	2	A	B	2	
B	1	2	A	C	2	
C	2	4	A	D	2	5
D	2	3	B	E	4	10
E	1	1	C	E	3	
			D	E	3	
			D	C		5

Fig. 3: An example of a set of operations and constraints.

tiveness of several heuristics for finding short schedules while still keeping the algorithm exact. A related technique [3] handles only exact delays and does not attempt to find short schedules. The scheduling method presented here can handle a more realistic model with delay intervals, and experimental results show that our techniques are competitive and practical.

II Problem Formulation

A Problem Statement

The problem input consists of a set of operations to be scheduled and a set of timing constraints to be satisfied by the schedule. An operation p is a sequence of instructions and has a range of execution delays $[\delta_l(p), \delta_u(p)]$. The delay values are non-negative and finite. Each operation may be a basic block or a subroutine containing branches internally, as long as the entry point and the exit point are unique. Any pair of operations can be related by a min or a max timing constraint or both between their start times. An example of this is shown in Fig. 3.

The output is an *interval schedule*, a complete ordering of the operations and the spacing between every adjacent pair. An interval schedule has the form $S = (p_1, I_S(p_1), p_2, I_S(p_2), \ldots, p_m)$, where p_i denotes the i-th operation and $I_S(p_i)$ denotes the amount of idle spacing separating p_i and p_{i+1}. While the execution delay of the operations may vary, this idle spacing is fixed.

A *run* r is an assignment of actual delay values $\delta_r(p_i)$ to operations p_i. We define $\tau_{S,r}$ to be a function that yields the start time of an operation p_i for a run r according to a schedule S. Formally,

$$\tau_{S,r}(p_k) = \begin{cases} \tau_{S,r}(p_{k-1}) + \delta_r(p_{k-1}) & \\ \qquad\qquad + I_S(p_{k-1}) & \text{if } k > 1 \\ 0 & \text{if } k = 1 \end{cases} \qquad (1)$$

In other words, the start time of an operation is the completion time of its predecessor's idle period. We are interested in only *consistent* assignments of execution delays, *i.e.*, between the predicted upper and lower bounds. We say that a run r is consistent if $\delta_l(p_i) \le \delta_r(p_i) \le \delta_u(p_i)$ for all operations p_i. We say that $\tau_{S,r}$ is *valid* if it does not violate any min or max timing constraints, denoted by $\text{minsep}(p_i, p_j)$ and $\text{maxsep}(p_i, p_j)$. For all specified min and max constraints on (p_i, p_j),

$$\begin{aligned} \tau_{S,r}(p_i) + \text{minsep}(p_i, p_j) &\le \tau_{S,r}(p_j) \\ \tau_{S,r}(p_i) + \text{maxsep}(p_i, p_j) &\ge \tau_{S,r}(p_j) \end{aligned} \qquad (2)$$

We say that a schedule S is *r-valid* if and only if r is consistent and $\tau_{S,r}$ is valid. Finally, we say that S is *valid* if and only if for all consistent r, S is r-valid. In other words, for a valid interval schedule, any execution will satisfy all timing constraints, as long as the execution delays are within their upper and lower bounds.

As an example, a valid schedule for the constraints in Fig. 3 is (A, 1, B, 0, D, 0, C, 1, E). Fig. 4 shows three different runs of the same schedule. A *min-run* is one where all operations take their lower-bound execution delays; *i.e.*, it assigns $\delta_l(p_i)$ to p_i. Similarly, a *max-run* assigns $\delta_u(p_i)$ to p_i for all i. By definition, both the min-run and max-run are consistent. We write $\tau_{S,l}$ and $\tau_{S,u}$ to denote the start time functions of the min-run and max-run, respectively.

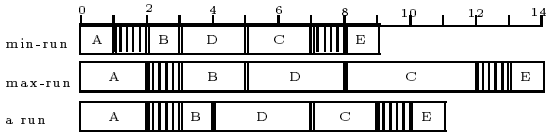

Fig. 4: Example runs of the schedule (A, 1, B, 0, D, 0, C, 1, E). The first row is the min-run. The second row is the max-run. The third row is another run with execution delays randomly chosen from the ranges. The idle spacing is the same in all runs.

B Graph Formulation

We formulate the problem in terms of a directed graph $G = (V, E, L, w, \delta, O)$, as an extension to the *constraint graph* in [4, 5]. The vertices, V, represent the operations, and the edges, E, represent constraints. The vertices and the edges are weighted. The vertex weights δ represent the execution delay of the operations, and the

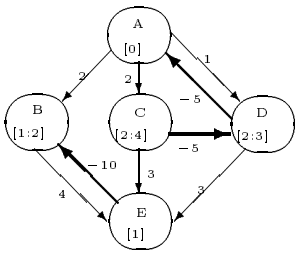

Fig. 5: Graph representation of the operations and constraints.

edge weights w represents the magnitude of the timing constraints. L and O are computed by the algorithm on successful return: L is an array that maps each operation to its start time in the min-run, and O is an ordered list of operations and represents their order in the schedule. L and O are used to construct the schedule S.

In the original formulation [5], the vertex weights are exact values; in this formulation, we generalize them to intervals. We call a constraint graph *simple* or *extended* based on whether it has simple or interval vertex weights. In our extended graph, each vertex p has an interval weight $[\delta_l(p), \delta_u(p)]$ for the lower and upper bounds on its execution delay.

Each edge $e = (p, q)$ in either an extended or a simple graph has a weight $w(e)$. For every edge (p, q) and all valid $\tau_{S,r}$, we require that $\tau_{S,r}(p) + w(p, q) \leq \tau_{S,r}(q)$. If $w(p, q)$ is non-negative, it is called a *forward edge*, and its weight represents a min constraint from the start of p to that of q. Otherwise, it is called a *backward edge*, and $-w(p, q)$ is a max constraint from q to p. Fig. 5 shows the graph representation of the operations and the constraints given in Fig. 3. Note that a forward edge (p, q) is a precedence relation and requires that p be ordered before q. A backward edge (*e.g.*, (C, D)), however, is not a precedence relation. Since each operation is executed without interruption, we assume that $w(p, q) \geq \delta_l(p)$ on all forward edges.

In this formulation, a positive cycle in the graph means certain constraints can never be satisfied and is called *ill-posed*. A *well-posed* graph is one that contains no positive cycles, and whose subgraph consisting of all its forward edges is acyclic and connected. We also assume there is an *anchor vertex* a from which all other vertices in the graph can be reached along the forward edges. A *feasible* graph is one for which a valid schedule exists. Note that all ill-posed graphs are infeasible, but not all well-posed graphs are feasible.

III Algorithm

The basic idea of the algorithm is as follows. It performs a topological traversal on the subgraph consisting of forward edges and selects one vertex at a time for serialization. It must ensure that appending the chosen vertex to the current partial schedule results in a valid schedule. The key idea is that if both the min-run and the max-run are valid, then all consistent runs (and therefore the schedule) are valid. See Theorem 1 in the Appendix section for the proof. If adding the selected vertex results in constraint violations, then algorithm backtracks to try another vertex, until all vertices are scheduled or exhausted. The algorithm is shown in Fig. 6.

```
Boolean
ScheduleInterval(extended graph G, anchor a, current vertex c)
{
    form G_l = (V[G], E[G], L[G], w[G]);
    if (not SingleSourceLongestPath(G_l, a))
       or not VerifyUpper(G, a)) return false;
    C := Candidates(c);
    if (C = ∅) return true;
    G' := G;
    while (C ≠ ∅) {
       p := SelectCandidate(C);
       foreach q ∈ C, p ≠ q {
             E[G'] := E[G'] ∪ (p, q) with weight δ_l(p)
       }
       append p to O[G'];
       if (ScheduleInterval(G', a, p)) return true;
       G := G'; // undo
    }
    return false;
}

Boolean
VerifyUpper(extended graph G, anchor a)
{
    foreach edge (p, q) ∈ E[G] {
       if ((p, q) are adjacent in O[G])
             W(p, q) := L[G](q) − L[G](p) + δ_u(p) − δ_l(p);
       else W(p, q) := w(p, q);
    }
    form G_u := (V[G], E[G], L', W);
    return SingleSourceLongestPath(G_u, a);
}
```

Fig. 6: The Interval Scheduling Algorithm

The algorithm is called with the constraint graph G, the anchor a, and the current vertex c. The anchor vertex a is the unique entry point to the graph. The algorithm builds a partial schedule at each recursive step, starting with the anchor a. The current vertex c is the last in this partial schedule and is initially set to a. Initially $O = (a)$ and $L(a) = 0$, $L(v \neq a) = -\infty$.

The first step of the algorithm is to check if the min-run for the current partial schedule is valid. This is done by calling the single-source longest path routine with $G_l = (V[G], E[G], L[G], w[G])$ [1]. The function SingleSourceLongestPath(G_l, a) computes the longest path lengths from the anchor a to all other vertices in the simple graph G_l, and returns TRUE if there are no positive cycles. Recall that having positive cycles in the

[1] The SingleSourceLongestPath() routine does not use vertex weights δ or the array O; therefore we omit them from the simple graph.

graph implies that some constraints can never be satisfied. The longest path lengths are returned in $L[G_l]$, which are the earliest possible start times for the operations in the min-run.

If the min-run for the current partial schedule is valid, the second step checks if the the max-run is valid by calling the subroutine VerifyUpper(). We induce the constraint graph G_u for the max-run by computing the edge weights W as follows:

$$W(p,q) := \begin{cases} L(q) - L(p) + \delta_u(p) - \delta_l(p) \\ \quad \text{if } p \text{ immediately precedes } q \text{ in } O \\ w(p,q) \quad \text{otherwise} \end{cases} \qquad (3)$$

If the max-run for the current partial schedule is valid, then the third step selects a new vertex and appends it to O. A vertex is a candidate if all of its predecessors as defined by forward edges in G have been scheduled (*i.e.* in O). If a candidate p is selected, it becomes a predecessor to all other candidates q, and we must either add a forward edge (p,q), or convert an existing backward edge (p,q) into a forward edge. The weight on this edge is assigned to be $\delta_l(p)$, because no computations may overlap. The scheduling routine is called recursively to schedule the rest of the graph, with p being the new "current vertex." If the choice of p results in an invalid schedule, then the algorithm undoes the edge addition and tries a different candidate.

When all vertices have been scheduled, the algorithm returns successfully with the complete ordering O for the vertices and the start times L, which then can be used to construct the schedule S. The array L is now exactly the same as the start time function $\tau_{S,l}$. The idle function is computed as follows:

$$I_S(p_i) = L(p_{i+1}) - L(p_i) - \delta_l(p_i), \quad i = 1 \ldots m-1. \quad (4)$$

The worst case complexity of this problem is NP-Hard, as are many scheduling problems. This algorithm solves "Sequencing with Release Times and Deadlines," , which is NP-complete in the strong sense [6].

IV Scheduling Heuristics

The algorithm always finds a valid schedule whenever one exists. However, when several valid schedules exist, it returns the first one it finds; it does not attempt to meet any other objective, such as minimizing schedule length. We would like to include schedule length as an additional objective, because a shorter schedule means potentially better processor utilization and smaller code size. Since the worst case complexity of finding a minimum-length valid schedule is exponential, we satisfy the length objective heuristically.

A *Heuristic Functions*

We have experimented with heuristics based on *slop*, *min-separation*, *in-degree*, some combinations, and Gupta's heuristic [3]. They are summarized in Fig. 7, where p is the "current vertex," and q is a candidate. All except the last are applied in the SelectCandidate() step.

Heuristic Ordering Function	
slop(q)	min $\delta_u(q) - \delta_l(q)$
minsep(p, q)	min $w(p,q)$ for forward edges (p,q)
indegree(q)	max# of incoming edges to q
Combined Heuristic Ordering	
slop/minsep	least slop; minsep as the tie-breaker
minsep/slop	minsep; least slop as the tie-breaker
minsep/$\delta_l(q)$	minsep; least $\delta_l(q)$ as tie-breaker
Scheduling Heuristic	
Gupta[3]	urgency, vertex elimination

Benchmark	#v	#fwd.e.	#back e.
rp-mode	8	12	1
fwd-mode	16	21	9
r-init	7	10	1
record	34	40	10
play	6	7	3
reset	6	6	2

Fig. 7: Heuristics and Examples Tested. p = current vertex, q = candidate

slop(q) : the difference between the upper and lower bound execution delays of a candidate q. We try to keep slop small because if a slop is larger than the corresponding *slack*, *i.e.*, difference between the max and min constraints, then no valid schedule exists.

minsep(p,q) : the min-constraint between the current vertex p and the candidate q, namely $w(p,q)$. A candidate with a smaller minsep require less idle time (locally) to meet its min constraint and is selected first.

indegree(q) : the number of incoming edges to a candidate q. A vertex with a high indegree may be due two things: either there are many specified constraints on it, or it has been eligible for a long time but has not been selected ("starvation"). This heuristic selects the candidate with the highest in-degree first.

Gupta's heuristic: a non-backtracking heuristic that uses *urgency* (max constraint) as its cost function. It is designed to find a valid schedule quickly using a *vertex elimination* scheme, which deletes those vertices that have been scheduled and their associated edges after re-expressing the constraints to be relative to the anchor. Note that this is a *heuristic*, which, unlike our *heuristic-ordering*, may fail to find a schedule when one exists.

B *Benchmarks*

The benchmarks include three code fragments of a robot controller and three fragments of a voice digitizer. All except the first benchmark have non-zero slop. The constraint topologies of these benchmarks can be classified into *parallel chains*, *tightly-synchronized clusters*, and *partially-ordered clusters*, where a *cluster* is a maximally strongly connected component in a constraint graph.

In parallel chains, several independent sequences of operations with their own constraints are to be interleaved. One such example is the "record" benchmark, which is

an unrolled loop that services three devices concurrently, and they synchronize only at the end.

In a tightly-synchronized cluster and partially-ordered clusters, the concurrent operations synchronize often, instead of only once at the end. The former contains max constraints across synchronization points, as exhibited by the "play" and "fwd-mode" benchmarks. Partially-ordered clusters, on the other hand, have only min constraints across synchronization points.

C Results and Analysis

We have implemented the heuristics in C++ and ran the benchmarks on a Sparcstation 2. We show the length bounds on the schedule produced by the algorithms in Fig. 8, with the best schedule length for each benchmark shown in bold. The run-time in milliseconds is shown in Fig. 9. Two cases timed out after 20 minutes. Overall, the heuristic with the best performance is minsep/slop, which considers candidates in order of increasing min-separation and uses slop as the tie-breaker. In this subsection, we discuss the effectiveness of the techniques on the different topologies.

Heuristics based on least-slop as the primary function do well in the tightly-synchronized clusters (fwd-mode and play) with nonzero slop. In this case, it is important to minimize the slop because the constraints for the entire graph are all related. On the other hand, Least-slop yields average results in partially-ordered clusters, because the (max) constraints do not go between clusters and thus slop does not matter as much. Least-slop times out in the parallel chains benchmark (record) due to starvation: one chain consistently contains operations with larger slop than the others. Since there are no precedence constraints between the chains, its operations are deferred until the very end. In this case, it would take an exponential number of steps to move the operations into valid positions, and this did not happen within 20 minutes. The in-degree heuristic prevents the problem of starvation by scheduling the candidate with the largest in-degree. However, this seems to be its only feature.

Combined heuristics using minsep as the primary function both perform well for the three constraint topologies. In parallel chains, they try to meet the min constraint in one chain by interleaving an operation from another chain. This is good for schedule length and avoids the starvation problem. For both the tightly coupled clusters and partially-ordered clusters, minsep-based heuristics greedily schedule the operations that result in the least idle time, regardless of the clusters they belong to. This enables them to find potentially shorter schedules than those techniques that schedule each cluster separately.

Gupta's heuristic does well in two partially-ordered clusters benchmarks (rp-mode and r-init), but fails on the parallel chains (record) and one of the tightly-synchronized clusters (play). This heuristic is based on the assumption that constraint interaction is somewhat localized or bounded. It works well for partially-ordered clusters because the max constraints can be decoupled between different clusters. When the constraints span a larger scope or have more interaction than can be predicted by the vertex elimination scheme, then the exact algorithm must be used.

V Conclusion and Future Work

We have presented a new technique for statically scheduling a set of operations with delay intervals. The output is a sequence of operations separated by idle instructions. The resulting schedule is guaranteed to meet all fine-grained min/max timing constraints for all combinations of actual delay values, without using any run-time executive. This is particularly important for low-level I/O operations in hardware-software co-synthesis.

We have also presented heuristic ordering functions for finding short schedules quickly. These are used by the exact algorithm to consistently produce short schedules, including cases where [3] did not find solutions. Some of our heuristics such as least-slop are specific to the interval scheduling problems, but most heuristics are readily applicable to previous techniques that did not try to optimize for schedule length [4, 5].

We have assumed the execution delays are independent of each other, and that all combinations of delay values within the given bounds are possible. In practice, the delays may be causally related, and that our scheduler may be too pessimistic about the worst-case delays because certain combinations may not be possible. For future work, we would like to incorporate additional information derived by the timing prediction program to enable us to find more valid and possibly shorter schedules.

Acknowledgment

We thank Elizabeth Walkup and Henrik Hulgaard for their helpful input.

References

[1] C. Y. Park. *Predicting Deterministic Execution Times of Real-Time Programs.* PhD thesis, Univ. of Washington, 1992. Tech. Report 92-08-02, Dept. of Computer Science & Engineering.

[2] E. A. Lee and S. Ha. Scheduling strategies for multiprocessor real-time DSP. In *Proc. of GLOBECOM*, volume 2, pages 1279–1283, Nov. 1989.

[3] R. K. Gupta and G. De Micheli. Constrained software generation for hardware-software systems. In *Proc. Third International Workshop on Hardware/Software Codesign*, pages 56–63, Sept. 1994.

Bench-mark	no heuristic	least slop		minsep			in-degree	Gupta's Heuristic
			minsep		slop	δ_l		
rp-mode	40	36	**32**	**32**	**32**	**32**	40	**32**
fwd-mode	57/135	**54/132**	**54/132**	57/135	**54/132**	**54/132**	57/135	57/135
r-init	38/54	37/53	37/53	38/54	**34/50**	**34/50**	38/54	**34/50**
record	172/269	timeout	timeout	173/270	**164/261**	167/264	173/270	fail
play	**10/16**	**10/16**	**10/16**	11/17	**10/16**	**10/16**	**10/16**	fail
reset	20/32	20/32	20/32	20/32	20/32	20/32	**19/31**	20/32

Fig. 8: Resulting scheduling lengths. Timeout after 20 minutes.

Bench-mark	no heu.	least slop	minsep				in-deg.	Gupta's Heuristic
			minsep		slop	δ_l		
rp-mode	12	15	14	14	14	14	19	12
fwd-mode	88	97	88	93	90	92	94	53
r-init	7	10	11	9	9	9	9	9
record	826	–	–	1307	756	729	965	143 (fail)
play	6	6	6	8	7	9	6	8 (fail)
reset	18	8	8	27	20	16	16	11

Fig. 9: Run-time (in ms) on a Sparcstation 2

[4] D. C. Ku and G. De Micheli. Relative scheduling under timing constraints: algorithms for high-level synthesis of digital circuits. *IEEE Trans. on Computer-Aided Design*, 11(6):696–717, June 1992.

[5] P. Chou and G. Borriello. Software scheduling in the co-synthesis of reactive real-time systems. In *Proc. 31st DAC*, pages 1–4, June 1994.

[6] M. R. Garey and D. S. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

Appendix: Proof

To prove the correctness of the algorithm, we need to show (1) ScheduleInterval() always halts. (2) If ScheduleInterval() returns FALSE then no feasible schedule exists. (3) If ScheduleInterval() returns TRUE then the schedule as constructed from O is valid.

(1) and (2) should be obvious, because the algorithm considers all possible orderings, which are finite. We prove (3) inductively on the length of O. By inductive hypothesis, $O = (p_1, \ldots, p_{k-1})$ for $k \geq 1$ is a valid ordering. It holds for $k = 1$ because $O = (a)$, which is correctly ordered by definition. Assume $(p_1, \ldots, p_k)$ is a prefix of a valid schedule S. In the algorithm, p_k is appended to O only if both SingleSourceLongestPath(G_l) and VerifyUpper(G_u) return TRUE.

Claim 1: SingleSourceLongestPath(G_l) either returns TRUE and assigns $L := \tau_{S,l}$ or returns FALSE if $\tau_{S,l}$ is not valid.

Lemma 1: VerifyUpper(G_u) returns TRUE iff $\tau_{S,u}$ is valid.

Proof: We show that the constraints in G_u as induced by VerifyUpper() are necessary and sufficient for the max-run. G_u contains all the necessary constraints of G_l. because $W(e) \geq w(e)$ for forward edges and $W(e) = w(e)$ for backward edges. To show sufficiency, we show that $\tau_{S,u}(p_j) - \tau_{S,u}(p_i) \geq W(p_i, p_j)$ for all consecutive pairs of operations p_i, p_{i+1}: $\tau_{S,u}(p_{i+1}) - \tau_{S,u}(p_i) = \delta_u(p_i) + I_S(p_i)$. $I_S(p_i)$ must be the same for all runs and is equal to $L(p_{i+1}) - L(p_i) - \delta_l(p_i)$. By substitution of $I_S(p_i)$, we get $\tau_{S,u}(p_j) - \tau_{S,u}(p_i) = L(p_{i+1}) - L(p_i) + \delta_u(p_i) - \delta_l(p_i) = W(p_i, p_j)$. □

Theorem 1: If $\tau_{S,l}$ and $\tau_{S,u}$ are valid then S is valid. That is, if both the min-run and the max-run of a schedule S are valid, then the schedule S is valid.

Proof: By definition, a schedule S is valid iff all consistent runs of S are valid. We need to show that if $\tau_{S,u}$ and $\tau_{S,l}$ are valid then any consistent $\tau_{S,r}$ is valid. In a consistent run r, the start time of p_j relative to that of p_i, where $i < j$, is $\tau_{S,r}(p_j) - \tau_{S,r}(p_i) = \sum_{k=i}^{j-1} \delta(p_k) + I_S(p_k)$. Constraint satisfaction has the form $\text{minsep}(p_i, p_j) \leq \tau_{S,r}(p_j) - \tau_{S,r}(p_i) \leq \text{maxsep}(p_i, p_j)$. We have

$$
\begin{array}{lll}
\text{minsep}(p_i, p_j) & & \\
\leq \tau_{S,l}(p_j) - \tau_{S,l}(p_i) & = & \sum_{k=i}^{j-1} \delta_l(p_k) + I_S(p_k) \\
\leq \tau_{S,r}(p_j) - \tau_{S,r}(p_i) & = & \sum_{k=1}^{j-1} \delta(p_k) + I_S(p_k) \\
\leq \tau_{u,r}(p_j) - \tau_{u,r}(p_i) & = & \sum_{k=1}^{j-1} \delta_u(p_k) + I_S(p_k) \\
\leq \text{maxsep}(p_i, p_j). \quad \square & &
\end{array}
$$

Theorem 2: If ScheduleInterval() returns TRUE then S as constructed from O is valid.

Proof: When the algorithm returns TRUE, S has satisfied both SingleSourceLongestPath() and VerifyUpper(). By Claim 1 and Lemma 1, $\tau_{S,u}$ and $\tau_{S,l}$ are valid, and by Theorem 1, S is valid. □